PDF Library SDK

Version 4.5

User Manual

Contact: pdfsupport@pdf-tools.com

Owner: **PDF Tools AG**

Kasernenstrasse 1
8184 Bachenbülach
Switzerland

www.pdf-tools.com

Table of Contents

1 Introduction

The PDF library originates from a development in early 1995. The library was designed to satisfy the requirements of the former Xerox DPP product, later called XDA (Xerox Document Assembly). Since then, more and more functionality has been added to the library. It constitutes the core of several own products and has been embedded into various third party products.

The basic functionality of the PDF library is to read in data from PDF files, present them in structured objects, and create new PDF files where such objects can be written to. The PDF library models the contents of a PDF file by C++ classes. You may want to read Adobe's PDF specification to gain the necessary background.

The PDF Library SDK supports PDF versions 1.1 which relates to Adobe Acrobat 2.1 up to 1.6 that comes with Adobe Acrobat 7.0.

2 Overview

The core classes of the PDF library comprise *PDFile* that encapsulates a PDF file and *PDObj*, which models an object in the PDF file.

The content of PDF objects is reflected by a hierarchically composed value (*PDValue*). A value can be a dictionary (*PDDictionary*), an object reference, or a another type like string or number. Dictionaries are collections of keys and associated values. Some objects have a data stream that belongs to them. This data is also attached to an object of class PDObj.

The library contains auxiliary classes to implement input from PDF files (PDParse, PDScan); they should be of no interest to a user of the library.

The basic functionality provided with PDFile and PDObj (file pdfile.h) is extended by derived classes. PDFInput is derived from PDFile with enhancements for basically two issues: copying pages to a PDF output file, and cashing objects in memory. Reading and writing of PDF files from/to memory is also supported.

PDFOutput is also derived from PDFile, but designed for enhancements that apply to output to a PDF file.

Note that the PDF library does not permit input and output at the same time to the same file. There is no updating of existing files, as the PDF standard would permit. A file that is written to is always created from scratch.

PDPage is a class derived from PDObj that models more precisely the behaviour of page objects. It is related to PDFInput, since PDFInput requires objects to be of this class for the CopyTo functionality. PDPage several enhancements over PDObj like adding contents, annotations, fonts or XObjects. Retrieval of page related information items is also supported.

Support for transforming a page from an input file into an XObject that can be used for output is included in „pdxobj.h" through the classes PDXObj and PDXSource.

Outlines (i. e. bookmarks) can be constructed and added to an output file. This support is found in „pdoutln.h".

Streams are used to carry many different kinds of data, notably the contents of a page. If you need access to an encoded contents stream, or if you would like to place text on a page, you use the classes PDStream or PDPgStream (pdstream.h).

3 Core Classes

3.1 PDFile

The class PDFile models a PDF file that is either being read from, or one that is being written to. It is not possible to alter an existing PDF file on disk; neither is it possible to make any changes to an object once it has been written out to a (new) PDF file. The class declaration is located in the header file „pdfile.h".

Reading from a PDF File

Reading from a PDF file is performed with the following steps:

```
PDFile theFile;
PDObj theObject;
theFile.Open("acrobat.pdf");
theObject.Read(theFile, theFile.GetInfoId());
```

After declaring appropriate variables, you gain access to information in the PDF file by first opening the file and then read from it by using the Read method (that belongs to the object in this sample here). The Read method fills in the data of „theObject".

An alternate method to read data from the file is using the ReadObj method of PDFile:

```
PDObj *pObj = theFile.ReadObj(1);
```

When you use ReadObj, a new object is dynamically created and returned to you with the data filled in. Note that this sample carries some dangers: we ask for object with id 1, but this object may not exist unless we have good reasons to believe this. ReadObj would return a NULL pointer in this case.

Please refer to the description of the PDObj class below for more information on gaining access to information within an object.

The ReadPages method can be used to traverse the pages tree of a PDF file. On traversal of the pages tree, OnReadPages is called; when a page is encountered, OnReadPage is called. The "pdls" sample shows how these methods can be overridden to add functionality.

Generally, page numbering starts at zero. This applies e. g. whenever a page is referred to by its number, as in link annotations. The member m_curPage counts page numbers before OnReadPage is called. Therefore, m_curPage contains the number of pages encountered so far and starts at one rather than zero.

Writing to a PDF file

PDF files can be written to in a variety of different ways. Be careful to obey the Adobe standards, it is easy to write messy files. The PDF Library SDK does not care much about the semantics of objects!

The creation of a PDF file happens according to the following scheme:

```
PDFile theFile;
theFile.Create("newfile.pdf");
theFile.Write("%comments are allowed");
theFile.WriteLn();
OBJID id = theFile.CreateObj();
theFile.WriteObjHeader(id);
theFile.Write(...);
theFile.WriteEndObj();
...
theFile.Close();
```

The Write method is overloaded to accept several parameter types: PDObj, CString, char*, numbers, PDValue, PDDictionary, arrays of bytes. WriteRef writes an object reference, WritePageRef writes an object reference to a page.

A PDObj is usually written to a file after reading it from another file and eventually modifying it. In this case, think about the id of this object: most of the time, it will not be the id it carries in the input file. If it is not related to anything you have written or are going to write, you must give it a new identification using the CreateObj method. It should not contain any object references inside.

If it is related to other objects that come from the same input file, i. e. if it is referenced from such objects or refers itself to such objects, you want to use the id „adoption" mechanism supplied in the PDF library. You have to replace the object id and all references it contains using the Adopted method. The PDCopyObj class helps you to do this for a whole hierarchy of objects.

Id adoption is a feature that maps object ids from a particular id scope - that of a chosen input file - to the scope of the output file. Whenever you choose a new input scope, you do this by a call to the ReserveIds method of the output file. It is not possible to save a mapping and restore it again, for example to merge pages of two input files. However, you can insert objects (pages) programmatically by using the CreateObj method that reserves new object ids.

Strings, numbers, PDValue and PDDictionary objects are written when you compose new objects as in the sample code above. PDF string values deserve your special attention; they are enclosed in left and right parentheses. If the text contains special characters - among them parenthesis - it has to be encoded appropriately. For this purpose, the PDF library supplies the functions MakePDFString and DecodePDFString (in pdfile.h).

Memory based Input/Output

The PDF Library SDK supports also reading or writing PDF files from/to a memory buffer.

If you choose for example to store a PDF file as a blob in a database, you can retrieve it to a memory buffer and open it using PDFile::MemOpen. An other use case is when you prefer to work with memory mapped files.

A web server application may not want to create the PDF file in the file system, but pipe the PDF file in response to a CGI or servlet request back to the browser. In this case, the output can be generated into a memory buffer by using the PDFile::MemCreate function. Note that you must Close the file to complete the output buffer. After that, you can use MemBuffer() and MemLength() to refer to the output buffer. The space for the output buffer is managed by the PDFile object and will be freed in the destructor of the object.

Standard Security Support

Support for standard security based on the encryption technique described in the Adobe PDF specifications is **optional**. This means that the API calls are present, but only functional with the corresponding code module contained in the library.

The functionality dealing with security is encapsulated in the classes PDFile and PDObj.

The PDFile::SetUserPassword and PDFile::SetOwnerPassword methods are used to provide password information after opening (or creating) a PDF file. The security flags are accessed via PDFile::PermissionFlags.

Since string and stream output is encrypted in secured files, you have to use the specific methods designed for these data types. PDFile::WriteEncoded will encrypt the data and then encode it. If you have used previous versions of the PDF library, you will have to replace calls like PDFile::WriteString("(some string data)") by PDFile::WriteEncoded("some string data");

PDF data is usually read via a PDObj object. This class has methods to facilitate encryption (for output) and decryption (for input), such as

- DecodeString, EncodeString
- DecryptStream, EncryptStream
- DecryptValue, EncryptValue

The data of a PDObj can be either decrypted (plain text) or encrypted, and care should be taken not to confuse these states. The PDObj::Read method will read in the data from the file and leave it encrypted. All other methods providing PDObj (or PDPage) objects will automatically decrypt the data. The PDObj::Write method will automatically encrypt the data.

Methods and Attributes

The class definition of PDFile is located in the file pdfile.h. It contains comments for the methods and attributes that may be of interest to an application programmer.

The destructor of PDFile takes care to free any dynamic memory associated with the PDFile object (m_template, closing the file to free the file handle, m_idMap, m_index,

m_parent, m_threadArr).

The close method frees m_parent, m_idMap, file handle, m_index, m_threadArr.

3.2 PDObj

Everything contained in a PDF file except header and trailer is a hierarchy of objects. The origin of all objects is the root object. PDObj objects carry their object id in the m_id attribute. The information contained in the object is stored in the „value" part (a protected attribute that you access using „AttrVal()"). Some objects have stream data; this data is attached to the value attribute (see PDValue below).

The class PDObj encapsulates all kinds of these objects. It discerns two specific types of objects that make up the pages of the document; the other object types are handled generically.

The type of an object is stored in the „m_kind" attribute. This attribute is actually determined from the value of the object (according to the /Type entry in the dictionary). Setting m_kind has no effect, it is just an indication for the efficient traversal of the pages tree.

3.3 PDValue

The PDValue class models all possible variants of simple or aggregated data that makes up the information contained in an object - at the root level or contained in an aggregate part of it.

The basic data types are object references, names, numbers, and strings. An object reference is something like „1 0 R", a name is e. g. „/Page" (in a dictionary like << /Type /Page >>), a number is an integer number as in << /Length 59 >>, an a string example is << /Title (De bello gallico) /Author (Julius Caesar) >>. The numerical data is stored in the m_num attribute, but also as string in m_string.

Aggregate types are arrays and dictionaries. Arrays are implemented as linked lists of PDValue objects, using the m_nextEl attribute. The m_num attribute of the array object contains the number of elements in the array. Note that array elements can be any basic data type or a dictionary. Starting with V1.4, arrays elements can also be arrays. In this case, make sure to use the access methods (GetFirstEl, GetNextEl). The behaviour with respect to the member variable m_nextEl has been preserved for compatibility with earlier versions of the library.

For a description of dictionaries, please refer to the next section.

Instances of the class PDValue can store a PDF stream, e. g. in the case of /Contents objects. In this case, they contain a dictionary which itself contains a /Length key and possibly /Filter keys. To construct such a class instance, you can use the method AssignStream. This method will automatically set the /Length key in the dictionary. (Make sure m_dict has been initialised before). It does not set or remove any encoding entries in the dictionary. Make sure these entries are set corresponding to the contents of the stream that you assign.

3.4 PDDictionary

Dictionaries are an aggregation of keys and associated values. Some common keys are predefined in the PDF library; in general, there is no limitation to keys, and the library handles this dynamically.

To gain access to the value associated e. g. with the /Length key, you would use either

PDDictionary *pDict = ...;

PDValue *pVal = pDict->GetAttrVal(PDDictionary::aLength);

or

PDValue *pVal = pDict->GetAttrVal("/Length");

To add another entry to an existing dictionary, you write the following code:

pDict->SaveAttrVal("/Author", pVal);

Keys are unique in a dictionary; if you apply SaveAttrVal to a dictionary with a key that already exists, the previous value is deleted and the new value is stored. Note that the value pointer that you pass is stored in the dictionary, and that the dictionary objects receives control over the value object. Before storing a value, you must allocate it using the „new" operator, and you may not delete it any more. You can delete the dictionary object, and this will automatically delete any values stored in it.

The DeleteAttr method deletes an entry from a dictionary. ChangeName allows you to change a specific key in the dictionary - this is more efficient than deleting and adding it again (you will hardly need this feature; it is used in one special case in the PDF library).

To traverse all keys and corresponding values in a dictionary, you use GetVal. The fpPos parameter works like an index, it starts at 0. GetVal returns FALSE (0) if the index runs out of range.

3.5 PDFInput

The main purpose of the class PDFInput is to selectively copy pages from the input file to an output file. It allows the modification of the pages on the fly. This is supported with an object cache that is also incorporated into PDFInput. Objects can be acquired selectively for alteration before the standard copy routine handles the page. During copy, the objects that are kept in the cache are used (rather than the original ones that would be read into memory from the input file).

The declarations for PDFInput are located in the header file „pdpage.h".

The CopyTo method works in conjunction with ReadPages, OnReadPage and OnReadPages. The latter methods contain the code that actually deals with copying. This means that you cannot use PDFInput to simply traverse the pages tree of a file and NOT copy pages to another file. You can derive a class from PDFInput, where you override ReadPages, OnReadPage and OnReadPages.

The sample program "pdcat" uses PDFInput to copy pages while doing some modifications to them.

How does PDFInput work

PDFInput incorporates a cache of objects that have been read using its GetObj method.

GetObj first looks at the cache (implemented by m_objOnHold); if the object is there, a pointer to it is returned. Otherwise, the object is read from the file and stored in the cache - and the pointer is returned. PeekObj can be used to check the cache for an object without reading it from the file.

The cache can be flushed either by using the ReleaseAll method or by using the ReleaseObj method. ReleaseObj can either release only the object that is specified, or also any other objects that are referenced from this object. The reference chain stops when a /Page or /Pages object would be reached (following link annotations and /Parent links would result in unpredictable behaviour).

Copying works as follows: the method CopyTo initializes the state of the member variables of PDFInput such that the methods dealing with page traversal select the desired pages. The ReserveIds method of the output file is called to flush a potentially existing id mapping table and reserve space for the one to come. Since CopyTo can be called several times in sequence, the array indicating which objects already have been copied is cleared. If no object template has been stored, CopyTo installs a PDPage template.

ReadPages, OnReadPages and OnReadPage are the methods that are called to traverse the pages tree of the input file. When only part of the pages are copied, the pages tree is modified to contain only the desired part of the pages. To this end, PDFInput requires PDPage objects to be read, because it makes use of the RemoveKid method. This method modifies recursively the /Pages object on the way up to the pages root. This is possible because traversal starts at the root object and recursively goes down to the leafs of the tree. When a leaf or sub tree that has to be omitted is found, all nodes up to the root are present on the stack and are linked via the m_parent member of PDPage.

Please note that CopyTo requires objects to be of class PDPage (or something derived from that).

As an alternative to the CopyTo method, you can use "CopyFew". This method does not traverse the whole pages tree, but rather descends the tree to a random page (or some random pages) to copy it. CopyFew is therefore appropriate to extract some pages from a large document.

Please be aware of a conceptual problem when copying only a range of pages: it is possible that these pages contain link annotations which refer to pages that are not copied. It is up to the PostCopyPage method to remove such annotations. If the page contains form fields that should be copied, there is a possible problem of having more instance of that field on pages that are not copied. The AcroForm dictionary must be reconstructed therefore. This is not yet automatically supported by the PDF library.

3.6 PDFOutput

The class PDFOutput is a rather tiny extension of PDFile. It stores objects of class PDStoredObj until after all other objects have been written to the output file. By overriding the WriteContents method of PDFile, PDFOutput triggers at this moment the output of the stored objects.

You would use stored objects as a convenient way to remember objects you want to write to the PDF file for which you do not have everything ready. This is the case for link annotations to pages whose id is not known yet, if you want to use the id for the destination (which is the more efficient and also more safe than using the page

number).

3.7 PDPage

The class PDPage is derived from PDObj and incorporates functionality related to /Page or /Pages objects.

The following features are related to these objects:

- adding a content object (to add text or graphics to a page)
- removing an entry from the page's dictionary (e.g. to strip off the annotations)
- add an annotation to the page
- add a font to the page's resources (which is required if that font is used in a content of the page)
- add an /XObject to the page's resources
- find the object in the pages tree that contains the MediaBox definition that applies to a page
- get the rectangle of the media box that applies to a page
- set the media box rectangle of the page (add it if it is defined elsewhere, or change it)
- remember the parent object
- remove a page or sub tree of pages from a /Pages object

To obtain objects of class PDPage rather than PDObj, you must use the PDFInput(PDFOutput*) constructor unless you do a "CopyTo". The m_template member of PDFile cannot be set directly to a PDPage object - derive your own class to do this.

3.8 PDFont

To create a page content with text, you need to refer to a font declaration. The class PDFont which is an extension of PDObj provides this support for the built in fonts like Helvetica, Times or Courier.

A typical scenario for using PDFont is

PDFont font;

font.Create("/FX1", "/Helvetica");

font.Write(output_file);

In this sample, the object id for the font object is created during the Write method. An alternate way is to create an object id first and then pass it as third parameter to Create.

The SetEncoding methods permit to set one of the standard (built in) encodings or to set a user defined encoding by referring to another PDF object (<< /Type /Encoding /Differences [...] >>; s. txt2pdf sample).

The PDFont object can be deleted after Write. Reuse of the PDFont object to create and

write several fonts is discouraged.

3.9 PDCopyObj

The class PDCopyObj is a helper class that extends the base class PDAttrScan to support the copying of an object tree from an input file to an output file. It is used for example in the context of the CopyTo method of PDFInput to copy everything belonging to a page. In the sample (pdcat), there is an example where PDAttrScan is derived not only to do the copy job but also patch certain items on the fly.

3.10 PDAnnotIterator

The class PDAnnotIterator helps to retrieve annotations from pages in a convenient representation (a polymorphic object rather than a general PDValue tree).

Currently, the recognition of Text and Link annotations of subtypes GoToR and Launch is supported.

Each call to GetNextAnnotData retrieves an annotation and stores it in a dynamically created object according to the type of the annotation. Make sure to delete this object when it is no longer used.

3.11 PDAction and Subclasses

The PDF library supports a number of standard action classes, such as „GoToR" (navigate to another page of a PDF file), „Launch" (activate another application program), and „URI" (web links for internet browser navigation).

PDAction is an abstract base class, so you will never create objects of that class, but rather deal with one of the subclasses PDLaunchAction, PDGoToRAction or PDURIAction. Objects of this type are found in conjunction with Annotations or book marks (outlines).

You can retrieve action information from a link annotation object or an outline object using the „GetAction" method of class PDFInput. Note that you are responsible to free PDAction objects created this way to avoid memory leaks.

3.12 PDAnnot, PDAnnotData and Subclasses

There are two major types of annotations in PDF: „Text" and „Link". Link annotations consist of a variety of subtypes like „GoToR", „Launch", or „URI". The PDF library supports the recognition of these types and subtypes of annotations by parsing the PDF objects containing such annotations. There is also support for constructing annotations and place them on pages, while resolving forward references to pages that are not yet created.

Class PDAnnotData is the base class of all annotation types. PDAnnot serves to intermediately store annotation data to be written to a PDF file, once the references to linked pages can be resolved (which is when the output file is about to be closed).

So, you will obtain PDAnnotData from parsing an input file e.g. by using

PDAnnotIterator::GetNextAnnotData().

Objects of class PDAnnot have to be created by you. You will typically attach these annotation objects to a particular page using PDPage::AddAnnotation. To not call AddAnnotation more than once for a particular PDAnnot object.

3.13 PDOutln

There is support for outlines (or book marks) through the classes PDOutln, PDOutlineTree and PDOutlineNode (header file „pdoutln.h").

You can construct the outline tree using the AppendKid method which is overloaded to generate actions of one of the subtypes described above.

The method AppendTree moves a whole outlines tree from an input file to the output file.

3.14 PDXObj, PDXSource

These two classes provide the functionality to e. g. add a logo on pages of a PDF file. The PDXObj encapsulates the XObject to be placed in the new PDF to be written, and PDXSource contains the functionality to extract the information for the XObject from the page of a PDF file.

There is a number of issues in this context:

In PDF 1.1, XObjects were not allowed to refer themselves to XObjects. The method HasXObjects was useful to detect that problem. In PDF 1.2, this is no longer a restriction.

Until version 1.4 of the PDF library, the contents stream of the page where the XObject is retrieved from had to be uncompressed, because some modifications must be made to it. The method HasEncodedStreams was useful to detect that problem. With the current release of the PDF library, this restriction no longer applies (actually, only LZW and FlateDecode is supported, but we have never found any other compression types applied to contents streams).

XObjects must be given a name that is unique within the scope of the page resources. Potential conflicts may come from either XObjects contained in the logo file or from such objects already contained in the PDF file to be enhanced with the logo. It may not be easy to check all pages of that file first in order to determine a new unique name for the XObject.

To make an XObject visible (add it to the page of an input file to produce an output file), you have to add suitable directives to the contents stream. The sample programs pdxt and pdcat demonstrate how to do that.

When placing a logo, you can run into the problem that it is not visible when placed on the background. The reason for that is that either the visible part of the logo lies outside of the visible portion of the page, or the page content is not transparent. The page content coming from a scanner is never transparent and will hide the logo, but there are also authoring tools which invisibly place a white rectangle that will have the same effect.

On the other hand, the logo may come from a source with a white (non-transparent)

background that will hide everything when the logo is put in the foreground of the page. So, either set the bounding box for the logo in order to clip it to the part that actually shall cover the page, or make sure the logo is transparent.

3.15 PDStream

Object of class PDStream store stream data. The declaration is located in the header file „pdstream.h".

In a PDF file, Streams are used for different purposes, e. g. to store the text and graphic contents of pages, but also thumb nails or font data. The class PDStream has a close relation to the class PDStreamBuf. PDStreamBuf only takes care of buffering the data, while PDStream allows manipulation of the data. PDStream incorporation LZW decoding of compressed streams, but not LZW compression (because of patent protection).

With release 1.3, PDStream also supports flate (zlib) encoding and decoding.

You can construct a stream using the PutBytes method and write it to a PDF file using the Write or WriteStreamObj method. You may want to have a look at the txt2pdf sample program for this.

Please note that GetLength returns the length of the uncompressed stream. The only way to get the length of the compressed stream is on writing it to a file (because only then, the actual compression is done).

The method ReplaceFontName is useful to patch font references in a text stream.

3.16 PDPgStream

Class PDPgStream is an extension of PDStream with support for the construction of page contents streams. The declaration of this class is located in „pdstream.h".

When starting a new stream that should contain text, use the TextDefaults method to reset text related characteristics like gray level, character and word spacing.

When mixing text and graphics, you need to switch modes in a PDF stream. For this purpose, there are two methods, NeedNextMode and NeedDrawMode. The text related methods automatically call NeedTextMode, while graphics related methods call NeedDrawMode.

For an in depth description of the stream operators, refer to the Adobe PDF specification.

3.17 PDFontDict

This class makes font information accessible to text scanning in contents streams. The implementation knows about the following standard fonts:

/Helvetica, /Helvetica-Bold, /Times-Roman, /Times-Italic, /Times, /ZapfDingbats, /Symbol, /Arial, /Arial-Bold, /Courier

Other fonts contained in PDF files should contain a /Widths attribute. PDFontDict will retrieve font metrics from there.

3.18 PDTextState

This class stores state information from text scanning which is necessary to accurately compute the width of a text token.

3.19 PDTextToken

An object of the class PDTextToken contains the results from text scanning as performed by PDTextScanner (s. below).

It stores the text token (string), its position in standard PDF coordinates, the font size (which corresponds to the height of the token on the page), the width of the text token, and its orientation.

The orientation is relative to the coordinate system; if there is a /Rotate entry in the /Page dictionary, it differs from the visual orientation when the page is displayed. This can typically be the case when pages are printed in landscape format.

3.20 PDTextScanner

The class PDTextScanner permits you to find text tokens on a PDF page. The behaviour can be controlled to some extent via the method „BreakOnBlank".

The default behaviour is to provide tokens that consist of as many characters as can obviously be retrieved from the stream. Whenever there is a change in a font or a stream operator is found that sets the text pointer, the token ends.

When BreakOnBlank is set, tokens will be broken down into pieces whenever there is more space between tow characters than about a space's witdh.

You should preferably use the class constructor that accepts a PDPage* parameter, because PDTextScanner can then find the font information required. We have found PDF files that contain streams that are broken down over several contents objects. Parsing requires that these streams are concatenated again.

The sample program pdw demonstrates the use of these features.

4 Classes of "PDPTDoc" Module

The "PDPTDoc" module (file pdptdoc.*) contains the classes that make up the so called "Prep Tool Suite component" (PT). The main features of this module are content analysis, content assembly, and dealing with Acrobat form fields.

4.1 PTInputDoc

This class enhances the class PDFInput in several ways. It

- supports reference counting for COM support
- permits to add, modify or delete form fields

- gives access to various objects like fonts, page content, document and page attributes, etc.

PTInputDoc cooperates with the other classes of the module as described below.

4.2 PTPrintDoc

This class adds functionality to PDFOutput for

- page content construction (in cooperation with PTPrintPage)
- filling in form data
- copy pages from existing PDF files
- copy bookmarks from existing files
- add bookmarks and links
- creating image objects for placement in the document

For a more detailed description of the functionality, refer to the Prep Tool Suite User's Manual.

4.3 PTFontRsc

The PTFontRsc class represents a collection of font definitions for the purpose of importing from an existing PDF file and reuse during content construction of an output PDF file.

4.4 PTFontEntry

Fonts that are used in content construction are stored in a PTFontEntry object, which itself is a member of the PTFontRsc collection.

4.5 PTPrintPage

A PTPrintPage object represents a layer of page content. Usually, pages just contain one layer, but it may also be interesting to use additional layers with content that is put on top of several pages (logo, header, footer, page numbers, etc.)

The PTPrintPage class is derived from the core class PDPgStream. It adds functionality for font handling and some standard PDF stream object constructors.

4.6 PTAnnotStore

PTAnnotStore stores the annotations (links) that shall be added to PDF pages that are created. There is a separate store object for each output page.

4.7 PTPageDir

PTPageDir contains all the PTAnnotStore objects for each individual page.

4.8 PDEnhancedTextScanner

The class PDEnhancedTextScanner provides some additional features compared to PDTextScanner. Most important, it can determine the width of a piece of text depending on font and a variety of settings that affect its appearance.

5 Linearization

Linearization is implemented in basically two new classes: PDLInput and PDLOutput. The input class performs the analysis of an existing PDF file, while the output class handles the linearization specific output.

The linearization classes are extensions of the PDFile class.

The use of the linearization classes is demonstrated in the pdlin command line application.

Functional extensions are possible, but should be implemented very carefully. You can override the PDLOutput::OnWriteObj method to add (or suppress) the standard optimization features. These are

- Removal of dictionary entries in /Pages objects that have been copied to the /Page leafs
- Compression of uncompressed streams (based on presence of a /Filter entry in the dictionary)
- Removal of references to objects not stored in the PDF file

6 Sample Applications

The sample applications are actually very useful utilities that demonstrate the power of the PDF Library SDK.

Please note that these utilities are copyright protected. You can use them for your own purposes and you can copy parts of the code to incorporate it into your product that you develop with the PDF Library SDK. However, your product must be significantly different from these utilities, and you may not incorporate the utilities into your product unless you have obtained written permission from PDF Tools AG for this.

All of the utilities print out a usage message when run with no arguments.

6.1 pdls

The pdls utility lists information about the pages tree of a PDF file. It can also print out the contents streams of the file.

6.2 pdinfo

The pdinfo program writes the entries of the info object and some important ids to standard output.

6.3 pdobj

The pdobj utility dumps the objects whose id is specified on the command line to

standard output. To find out the id of a particular page, you would first use pdls. When you specify a file name only, pdobj will print the info and Catalog objects.

When the option -s is specified, pdobj will print also stream contents.

6.4 pdcat

The pdcat utility demonstrates how a number of files can be concatenated to a single PDF file. This program can also add bookmarks related to each of the input files; it can even copy existing bookmarks from the input files into the output file.

The pdcat sample also demonstrates a simple manipulation of page contents. When the "clip" option is specified on the command line, the corresponding rectangle is clipped on each page (actually only on the first content of the page - but usually, there is only one content).

With release 1.4, pdcat now incorporates a lot more functionality. It can add a logo (see pdxt), but also add link annotations and bookmarks according to directives from a separate input file.

6.5 pdtoc

The pdtoc utility creates a PDF file that contains a page with a list of links to files specified on the command line. There are may options to control the behaviour, like bookmark copying, placing the creation date of the file onto the page, setting the page width, setting a title string on top of the page, and giving a document title to the new file.

pdcat and pdtoc can be used to build a contents document for a whole hierarchy of documents.

6.6 pdxt

The pdxt program demonstrates how a background logo can be added to some pages of a PDF document. The logo is converted into an XObject, and a content that refers to the XObject is added on the desired pages.

Its functionality is now also integrated in pdcat.

6.7 txt2pdf

The txt2pdf program demonstrates the creation of a PDF file based on ASCII text input. It uses PDPgStream to compose the contents stream.

6.8 pdw

This program demonstrates how text tokens can be retrieved from a contents stream along with some metrics information like position, size, and orientation.

6.9 pdwebl

The "pdwebl" program demonstrates how textual content analysis of an existing PDF file can be used to add internet links at the location of selected text pieces.

There are several issues that make this interesting:

Many applications that produce PDF create small fragments of text that must be reassembled. The re-assembly is based on heuristics of "geographical" placement. Use of multi-column text can make the correct text assembly very difficult.

pdwebl assembles the text of a line before matching is applied. If a pattern spans over the end of a line, it will not be recognized.

Often, it is desired that links are visualized in some way. Acrobat can add a border to the box that represents the link. This box is not visible on a printout. It is also possible to change the content of the page to reflect the presence of a link, e. g. by changing the color of the text, or by adding a line blow the text. All this requires a programming effort – and will affect the printout.

By the way: pdwebl also shows how memory based PDF files can be handled. Depending on the options settings, it reads from standard input into a memory buffer and passes this to the PDF library. Output can also be collected in a memory buffer – and then written to (e.g.) standard output.

6.10 pdsplit

The pdsplit program demonstrates how link annotations can be changed on the fly when splitting a PDF file into several output files.

This program has been developed to prepare PDF files for a web server application which counts access to individual pages of the PDF files.

7 Appendix

7.1 Things to observe

Security

PDF files can be encrypted to provide security features. The PDF Library SDK supports "Standard" PDF security as described in the Adobe PDF specifications.

Copying

PDFile and PDObj objects (and objects of derived classes) cannot be copied; the copy constructor is made private to prevent you from doing this. If you write functions that take PDFile parameters, pass these parameters by reference.

Memory Usage

Keeping many objects in memory requires heap space. Try to free objects that you do not need any more. If you have to process all pages of a file, use the recursive traversal of the ReadPages method. If you use PDFInput::GetObj, make sure to apply ReleaseObj or ReleaseAll if you are dealing with large files. When the files are always small, there is no problem.

Try to avoid memory leaks. Whenever you use a method that returns a pointer, make sure whose responsibility it is to free the data again. PDFInput::GetObj keeps the data in a cache, and you may not free the data yourself. On the other hand, when extracting annotation data from a page using PDAnnotIterator, this data is not cached by the PDF library, and it is your responsibility to free it.

Multithreading

The PDF library is thread safe in the sense that multiple threads are allowed to concurrently access distinct objects (files). It is also possible for the application to synchronize access to PDF objects between several threads.

Thread safety is not ensured for error output, however – which is by default disabled anyway.

Error Handling

When the PDF library encounters unexpected situations, it can print an error message to standard error or some file (s. PD_ERROR macro definition in „pdimpl.h"). Error output is controlled via the "pd_set_error_output" function (s. "pdimpl.h"). Error logging is not thread safe.

When an unexpected situation is encountered within functions that return a pointer result, NULL (0) is returned. This is also the case when the result is an OBJID, because

zero is not a valid object identification. When a PAGENR is returned, a value less than zero means an error, because 0 is a valid page number (page numbering starts at zero). In the context of a PDFile object, the error code "m_err" is set.

Compiling on MS Windows

As of V2.0, MSVC 1.52 i(WIN16) s not longer supported.

The binary release for Windows systems is compiled with MSVC 6.0. There are several variants how the library is built depending on

- whether it is used with or without MFC
- whether it is to be linked statically or as DLL
- whether it is to be used with the multithreaded Win32 libraries or not
- debug setting

When the PDF Library SDK is used together with MFC, the MFC implementation of CString is used. It is possible to use the PDF library without MFC and still have CString objects avaiable (as on UNIX platforms) based on the CString subset implemented in the PDF Library.

Release Library	Debug Library	Using MFC	Type	Encryption support	Thread model
PDAFX	PDAFXD	Yes	DLL	No	Multithreaded
PDLIB	-	No	Static	No	Single
PDAFXE	PDAFXD	Yes	DLL	Yes	Multithreaded
PDLIBE	-	No	Static	Yes	Single

If you have a source code license and want to compile the library with MSVC, the macroes _AFX and _AFXDLL will control whether CString comes from MFC or not.

The compiler macro _WINDLL will control whether "export" directives are generated to make the API classes available to the linker.

Using Different Compiler Settings

You may encounter problems when using special compiler options to build an application using the PDF library in binary form. There are some precautions for this when using MS Visual C++ and packing options.

However, there are cases where no simple solution exists. If the linker complains about missing functions that are inlines, the problem is probably that you are compiling with debugging option enabled but linking to a PDF library archive that was compiled with debugging off. So, make sure you use corresponding settings (check, if there is a debug version of the PDF library to link with in this case).

A problem that has been found when using PDAFX with MFC. CString objects may be passed between code with different DEBUG settings, resulting in access violations. This is probably due to different storage allocation of CString objects. Thus, make sure you are using the correct PDAFX(D) DLL.

7.2 Trouble shooting

Compilation with MSVC When Using MFC

Because of a strange feature (bug?) of MSVC, you cannot use precompiled headers when including „pdfile.h". The statement that causes troubles is „#ifdef _AFXDLL ... #include <afx.h>". You can edit pdfile.h and replace the whole ifdef part by #include <stdafx.h>. But be sure to use the AFX version of the library.

Text Operator Dependencies

Adobe introduced a new restriction on text operators with Version 3.01. In order to print correctly on postscript printers, the Tc and Tw operators must not be issued before a font has been set using Tf. The sample txt2pdf has been updated accordingly.

8 Index

9 Licensing